# 9 Classification

This topic returns to prediction. Unlike linear regression where we were predicting a numeric value, in this case we are predicting a class: winner or loser, yes or no, rich or poor, positive or negative. Ideas from linear regression can be applied here, but we will instead overview a different, but still beautiful family of techniques based on linear classification.

This is perhaps *the* central problem in data analysis. For instance, you may want to predict:

- will a sports team win a game?
- will a politician be elected?
- will someone like a movie?
- will someone click on an ad?
- will I get a job? (If you can build a good classifier, then probably yes!)

Each of these is typically solved by building a general purpose classifier (about sports or movies etc), then applying it to the person in question.

## 9.1 Linear Classifiers

Our input here is a point set $X \subset \mathbb{R}^d$, where each element $x_i \in X$ also has an associated label $y_i$. And $y_i \in \{-1, +1\}$.

Like in regression, our goal is prediction and generalization. We assume each $(x_i, y_i) \sim \mu$; that is, each data point pair, is drawn iid from some fixed but unknown distribution. Then our goal is a function $g : \mathbb{R}^d \to \mathbb{R}$, so that if $y_i = +1$, then $g(x_i) \geq 0$ and if $y_i = -1$, then $g(x_i) \leq 0$.

We will restrict that $g$ is linear. For a data point $x \in \mathbb{R}^d$, written $x = (x^{(1)}, x^{(2)}, \ldots, x^{(d)})$ we enforce that

$$g(x) = \alpha_0 + x^{(1)}\alpha_1 + x^{(2)}\alpha_2 + \ldots + x^{(d)}\alpha_d = \alpha_0 + \sum_{j=1}^{d} x^{(j)}\alpha_j,$$

for some set of scalar parameters $\alpha = (\alpha_0, \alpha_1, \alpha_2, \ldots, \alpha_d)$. Typically, different notation is used: we set $b = \alpha_0$ and $w = (w_1, w_2, \ldots, w_d) = (\alpha_1, \alpha_2, \ldots, \alpha_d) \in \mathbb{R}^d$. Then we write

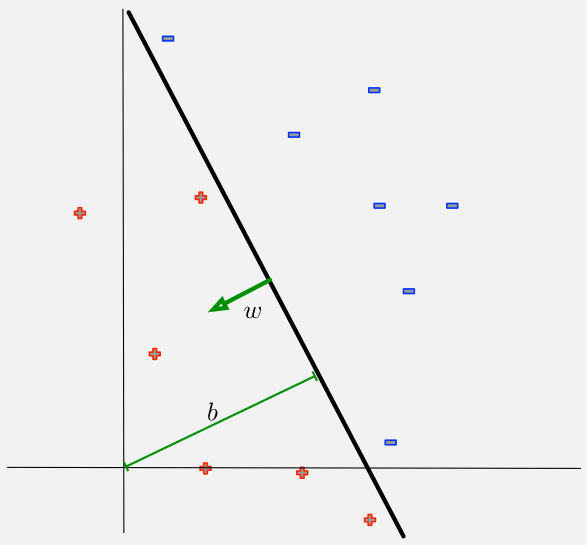
$$g(x) = b + x^{(1)}w_1 + x^{(2)}w_2 + \ldots + x^{(d)}w_d = \langle w, x\rangle + b.$$

We can now interpret $(w, b)$ as defining a halfspace in $\mathbb{R}^d$. Here $w$ is the normal of that halfspace boundary (the single direction orthogonal to it) and $b$ is the distance from the origin $\mathbf{0} = (0, 0, \ldots, 0)$ to the halfspace boundary in the direction $w/\|w\|$. Because $w$ is normal to the halfspace boundary, $b$ is also distance from the closest point on the halfspace boundary to the origin (in any direction).

We typically ultimately use $w$ as a unit vector, but it is not important since this can be adjusted by changing $b$. Let $w, b$ be the desired halfspace with $\|w\| = 1$. Now assume we have another $w', b'$ with $\|w'\| = \beta \neq 1$ and $w = w'/\|w'\|$, so they point in the same direction, and $b'$ set so that they define the same halfspace. This implies $b' = b/\beta$. So the normalization of $w$ can simply be done post-hoc without changing any structure.

Recall, our goal is $g(x) \geq 0$ if $y = +1$ and $g(x) \leq 0$ if $y = -1$. So if $x$ lies directly on the halfspace then $g(x) = 0$.

**Example: Linear Separator in $\mathbb{R}^2$**
Here we show a set $X \in \mathbb{R}^2$ of 13 points with 6 labeled $+$ and 7 labeled $-$. A linear classifier perfectly separates them. It is defined with a normal direction $w$ (pointing towards the positive points) and an offset $b$.



Using techniques we have already learned, we can immediately apply two approaches towards this problem.

**Linear classification via linear regression.** For each data points $(x_i, y_i) \in \mathbb{R}^d \times \mathbb{R}$, we can immediately represent $x_i$ as the value of $d$ explanatory variables, and $y_i$ as the single dependent variable. Then we can set up a $n \times (d+1)$ matrix $M$, where the $i$th row is $(1, x_i)$; that is the first coordinate is 1, and the next $d$ coordinates come from the vector $x_i$. Then with a $y \in \mathbb{R}^n$ vector, we can solve for

$$\alpha = (M^T M)^{-1} M^T y$$

we have a set of $d+1$ coefficients $\alpha = (\alpha_0, \alpha_1, \ldots, \alpha_d)$ describing a linear function $g : \mathbb{R}^d \to \mathbb{R}$ defined

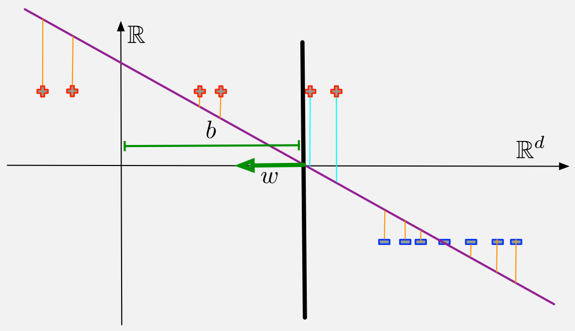
$$g(x) = \langle \alpha, (1, x) \rangle.$$

Hence $b = \alpha_0$ and $w = (\alpha_1, \alpha_2, \ldots, \alpha_d)$. For $x$ such that $g(x) > 0$, we predict $y = +1$ and for $g(x) < 0$, we predict $y = -1$.

However, this approach is optimizing the wrong problem. It is minimizing how close our predictions $g(x)$ is to $-1$ or $+1$, by minimizing the sum of squared errors. But our goal is to minimize the number of mispredicted values, not the numerical value.

**Example: Linear Regression for Classification**

We show 6 positive points and 7 negative points in $\mathbb{R}^d$ mapped to $\mathbb{R}^{d+1}$. All of the $d$-coordinates are mapped to the $x$-axis. The last coordinate is mapped to the $y$-axis and is either $+1$ (a positive points) or $-1$ (a negative points). Then the best linear regression fit is shown, and the points where it has $y$-coordinate $0$ defines the boundary of the halfspace. Note, despite there being a linear separator, this method misclassifies two points because it is optimizing the wrong measure.



**Linear classification via gradient descent.** Since, the linear regression SSE cost function is not the correct one, what is the correct one? We might define a cost function $\Delta$

$$\Delta(g,(X,y)) = \sum_{i=1}^{n}(1 - \mathbb{1}(\mathsf{sign}(y_i) = \mathsf{sign}(g(x_i)))$$

which uses the *identity function* $\mathbb{1}$ (defined $\mathbb{1}(\text{TRUE}) = 1$ and $\mathbb{1}(\text{FALSE}) = 0$) to represent the *number* of misclassified points. This is what we would like to minimize.

Unfortunately, this function is discrete, so it does not have a useful (or well-defined) derivative. And, it is also not convex. Thus, encoding $g$ as a $(d+1)$-dimensional parameter vector $(b, w) = \alpha$ and running gradient descent is not feasible.

However, most classification algorithms run some variant of gradient descent. To do so we will use a different cost function as a proxy for $\Delta$, called a *loss function*. We explain this next.

### 9.1.1 Loss Functions

To use gradient descent for classifier learning, we will use a proxy for $\Delta$ called a *loss* functions $\mathcal{L}$. These are sometimes implied to be convex, and their goal is to approximate $\Delta$. And in most cases, they are *decomposable*, so we can write

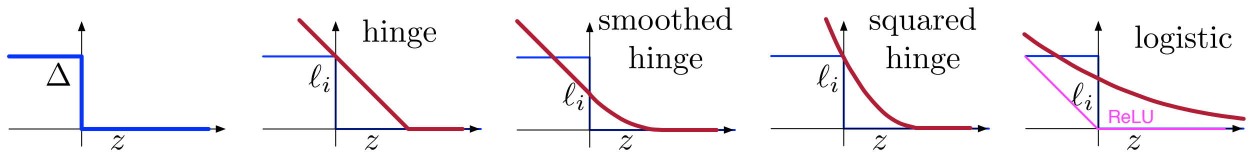
$$\begin{aligned}\mathcal{L}(g,(X,y)) &= \sum_{i=1}^{n} \ell_i(g,(x_i,y_i)) \\ &= \sum_{i=1}^{n} \ell_i(z_i) \text{ where } z_i = y_i g(x_i).\end{aligned}$$

Note that the clever expression $z_i = y_i g(x_i)$ handles when the function $g(x_i)$ correctly predicts the positive or negative example in the same way. If $y_i = +1$, and correctly $g(x_i) > 0$, then $z_i > 0$. On the other hand, if $y_i = -1$, and correctly $g(x_i) < 0$, then also $z_i > 0$. For instance, the desired cost function, $\Delta$ is written

$$\Delta(z) = \begin{cases} 0 & \text{if } z \geq 0 \\ 1 & \text{if } z < 0. \end{cases}$$

Most *loss functions* $\ell_i(z)$ which are convex proxies for $\Delta$ mainly focus on how to deal with the case $z_i < 0$ (or $z_i < 1$). The most common ones include:

- hinge loss: $\ell_i = \max(0, 1 - z)$
- smoothed hinge loss: $\ell_i = \begin{cases} 0 & \text{if } z \geq 1 \\ (1-z)^2/2 & \text{if } 0 < z < 1 \\ \frac{1}{2} - z & \text{if } z \leq 0 \end{cases}$
- squared hinge loss: $\ell_i = \max(0, 1 - z)^2$
- logistic loss: $\ell_i = \ln(1 + \exp(-z))$



The hinge loss is the closest convex function to $\Delta$; in fact it strictly upper bounds $\Delta$. However, it is non-differentiable at the "hinge-point," (at $z = 1$) so it takes some care to use it in gradient descent. The smoothed hinge loss and squared hinge loss are approximations to this which are differentiable everywhere. The squared hinge loss is quite sensitive to outliers (similar to SSE). The smoothed hinge loss (related to the Huber loss) is a nice combination of these two loss functions.

The logistic loss can be seen as a continuous approximation to the ReLU (rectified linear unit) loss function, which is the hinge loss shifted to have the hinge point at $z = 0$. The logistic loss also has easy-to-take derivatives (does not require case analysis) and is smooth everywhere. Minimizing this loss for classification is called *logistic regression*.

### 9.1.2 Cross Validation and Regularization

Ultimately, in running gradient descent for classification, one typically defines the overall cost function $f$ also using a regularization term $r(\alpha)$. For instance $r(\alpha) = \|\alpha\|^2$ is easy to use (has nice derivatives) and $r(\alpha) = \|\alpha\|_1$ (the $L_1$ norm) induces sparsity (for reasons not covered in this class). In general, the regularizer typically penalizes larger values of $\alpha$, resulting in some bias, but less over-fitting of the data.

The regularizer $r(\alpha)$ is combined with a loss function $\mathcal{L}(g_\alpha, (X, y)) = \sum_{i=1}^n \ell_i(g_\alpha, (x_i, y_i))$ as

$$f(\alpha) = \mathcal{L}(g_\alpha, (X, y)) + \eta r(\alpha),$$

where $\eta \in \mathbb{R}$ is a *regularization parameter* that controls how drastically to regularize the solution.

Note that this function $f(\alpha)$ is still decomposable, so one can use batch, incremental, or most commonly stochastic gradient descent.

**Cross-validation.** Backing up a bit, the *true* goal is not minimizing $f$ or $\mathcal{L}$, but predicting the class for new data points. For this, we again assume all data is drawn iid from some fixed but unknown distribution. To evaluate how well out results generalizes, we can use cross-validation (holding out some data from the training, and calculating the expected error of $\Delta$ on these help out "testing" data points).

We can also choose the regularization parameter $\eta$ by choosing the one that results in the best generalization on the test data after training using each on some training data.

## 9.2 Perceptron Algorithm

Of the above algorithms, generic linear regression is not solving the correct problem, and gradient descent methods do not really use any structure of the problem. In fact, we could have replaced the linear function $g_\alpha(x) = \langle \alpha, (1, x)\rangle$ with any function $g$ (even non-linear ones) as long as we can take the gradient.

Now we will introduce the *perceptron algorithm* which explicitly uses the linear structure of the problem. (Technically, it only uses the fact that there is an inner product – which we will exploit in generalizations.)

**Simplifications:** For simplicity, we will make several assumptions about the data. First we will assume that the best linear classifier $(w^*, b^*)$ defines a halfspace whose boundary passes through the origin. This implies $b^* = 0$, and we can ignore it. This is basically equivalent to (for data point $(x'_i, y_i) \in \mathbb{R}^{d'} \times \mathbb{R}$ using $x_i = (1, x'_i) \in \mathbb{R}^d$ where $d' + 1 = d$.

Second, we assume that for all data points $(x_i, y_i)$ that $\|x_i\| \leq 1$ (e.g., all data point live in a unit ball). This can be done by choosing the point $x_{\max} \in X$ with largest norm $\|x_{\max}\|$, and dividing all data points by $\|x_{\max}\|$ so that point has norm 1, and all other points have smaller norms.

Finally, we assume that there exists a perfect linear classifier. One that classifies each data point to the correct class. There are variants to deal with the cases without perfect classifiers, but which are beyond the scope of this class.

**The algorithm.** Now to run the algorithm, we start with some normal direction $w$ (initialized as any positive point), and then add mis-classified points to $w$ one at a time.

```
Algorithm 9.2.1 Perceptron($X$)
  Initialize $w = y_i x_i$ for any $(x_i, y_i) \in (X, y)$
  repeat
    For any $(x_i, y_i)$ such that $y_i \langle x_i, w\rangle < 0$ (is mis-classified) : update $w \leftarrow w + y_i x_i$
  until (no mis-classified points   or   $T$ steps)
  return $w \leftarrow w/\|w\|$
```

Basically, if we find a mis-classified point $(x_i, y_i)$ and $y_i = +1$, then we set $w = w + x_i$. This makes $w$ "point" more in the direction of $x_i$, but also makes it longer. Having $w$ more in the direction of $w$, tends to make it have dot-product (with a normalized version of $w$) closer to 1.

Similar, if we find a mis-classified point $(x_i, y_i)$ with $y_i = -1$, the we set $w = w - x_i$; this points the negative of $w$ more towards $x_i$, and thus $w$ more away from $x_i$, and thus its dot product more likely to be negative.

**The margin.** To understand why the perceptron works, we need to introduce the concept of a margin. Given a classifier $(w, b)$, the margin is

$$\gamma = \min_{(x_i, y_i) \in X} y_i(\langle w, x_i\rangle + b).$$

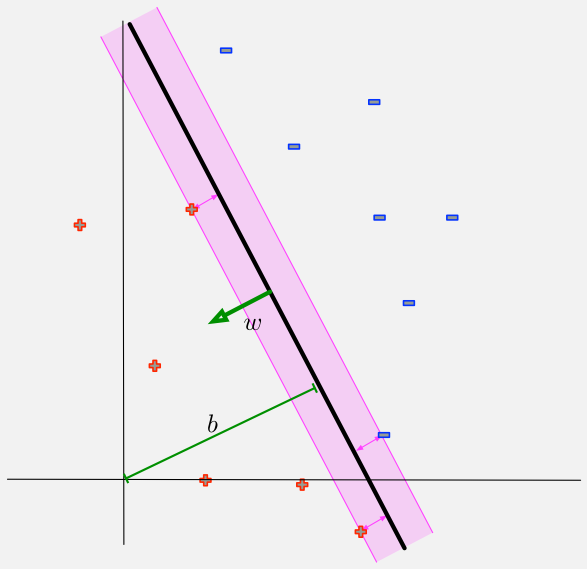
Its the minimum distance of any data point $x_i$ to the boundary of the halfspace. In this sense the optimal classifier (or the maximum margin classifier) $(w^*, b^*)$ is the one that maximizes the margin

$$(w^*, b^*) = \arg\max_{(w,b)} \min_{(x_i, y_i) \in X} y_i(\langle w, x_i\rangle + b)$$
$$\gamma^* = \min_{(x_i, y_i) \in X} y_i(\langle w^*, x_i\rangle + b^*).$$

A max-margin classifier, is one that not just classifies all points correctly, but does so with the most "margin for error." That is, if we perturbed any data point or the classifier itself, this is the classifier which can account for the most perturbation and still predict all points correctly. It also tends to *generalize* (in the cross-validation sense) to new data better than other perfect classifiers.

**Example: Margin of Linear Classifier**

For a set $X$ of 13 points in $\mathbb{R}^2$, and a linear classifier defined with $(w, b)$. We illustrate the margin in pink. The margin $\gamma = \min_{(x_i,y_i)} y_i(\langle w, x_i\rangle + b)$. The margin is drawn with an $\leftrightarrow$ for each support point.



The maximum margin classifier $(w^*, b^*)$ for $X \subset \mathbb{R}^d$ can always be defined uniquely by $d + 1$ points (at least one negative, and at least one positive). These points $S \subset X$ are such that for all $(x_i, y_i) \in S$

$$\gamma^* = y_i\langle w^*, x_i\rangle + b.$$

These are known as the *support points*, since they "support" the margin strip around the classifier boundary.

**Why perceptron works.** We claim that after at most $T = (1/\gamma^*)^2$ steps (where $\gamma^*$ is the margin of the maximum margin classifier), then there can be no more mis-classified points.

To show this we will bound two terms as a function of $t$, the number of mistakes found: $\langle w, w^*\rangle$ and $\|w\|^2 = \langle w, w\rangle$, before we ultimately normalize $w$ in the **return** step.

First we can argue that $\|w\|^2 \leq t$, since each step increases $\|w\|^2$ by at most 1:

$$\langle w + y_i x_i, w + y_i x_i\rangle = \langle w, w\rangle + (y_i)^2\langle x_i, x_i\rangle + 2y_i\langle w, x_i\rangle \leq \langle w, w\rangle + 1 + 0.$$

This is true since if $x_i$ is mis-classified, then $y_i\langle w, x_i\rangle$ is negative.

Second, we can argue that $\langle w, w^*\rangle \geq t\gamma^*$ since each step increases it by at least $\gamma^*$. Recall that $\|w^*\| = 1$

$$\langle w + y_i x_i, w^*\rangle = \langle w, w^*\rangle + (y_i)\langle x_i, w^*\rangle \geq \langle w, w^*\rangle + \gamma^*.$$

The inequality follows from the margin of each point being at least $\gamma^*$ with respect to the max-margin classifier $w^*$.

Combining these facts together we obtain

$$t\gamma^* \leq \langle w, w^*\rangle \leq \langle w, \frac{w}{\|w\|}\rangle = \|w\| \leq \sqrt{t}.$$

Solving for $t$ yields $t \leq (1/\gamma^*)^2$ as desired.

## 9.3 Kernels

It turns out all we need to get any of the above machinery to work is a well-defined (generalized) inner-product. For two vectors $p = (p_1, \ldots, p_d), q = (q_1, \ldots, q_d) \in \mathbb{R}^d$, we have always used as the inner product:

$$\langle p, q \rangle = \sum_{i=1}^{d} p_i \cdot q_i.$$

However, we can define inner products more generally as a kernel $K(p, q)$. For instance, we can use

- $K(p, q) = \exp(-\|p - q\|^2/\sigma^2)$ for the Gaussian kernel, with bandwidth $\sigma$,
- $K(p, q) = \exp(-\|p - q\|/\sigma)$ for the Laplace kernel, with bandwidth $\sigma$, and
- $K(p, q) = (\langle p, q \rangle + c)^r$ for the polynomial kernel of power $r$, with control parameter $c > 0$.

Then we define our classification function

$$g(x) = K(x, w) + b.$$

If $g(x) > 0$, we classify $x$ as positive, and if $g(x) < 0$, we classify $x$ as negative.

For gradient decent, again $\alpha = (\alpha_0, \alpha_1, \ldots, \alpha_d) = (b, w)$. We just need to take the gradient for each term of the loss function $\ell_i(z_i)$ for $z_i = y_i g(x_i)$ as before.

And for perceptron, we check $y_i K(x_i, w) > 0$ to see if $(x_i, y_i)$ is mis-classified, and still simply add $w \leftarrow w + y_i x_i$ as before.

**Are these linear classifers?** No. In fact, this is how you model various forms of non-linear classifiers. The "decision boundary" is no longer described by the boundary of a halfspace. For the polynomial kernel, the boundary must now be a polynomial surface of degree $r$. For the Gaussian and Laplace kernel it can be even more complex; the $\sigma$ parameter essentially bounds the curvature of the boundary.

## 9.4 $k$NN Classifiers

Now for something completely different. There are many ways to define a classifier, and we have just touched on some of them. These include decision trees (which basically just ask a series of yes/no questions and are very interpretable) to deep neural networks (which are more complex, far less interpretable, but can achieve more accuracy). We will describe one more simple classifier.

The $k$-NN classifier (or *$k$-nearest neighbors classifier*) works as follows. Choose a scalar parameter $k$ (it will be far simpler to choose $k$ as an odd number, say $k = 5$). Next define a *majority* function $\mathsf{maj} : \{-1, +1\}^k \to \{-1, +1\}$. For a set $Y = (y_1, y_2, \ldots, y_k) \in \{-1, +1\}^k$ it is defined

$$\mathsf{maj}(Y) = \begin{cases} +1 & \text{if more than } k/2 \text{ elements of } Y \text{ are } +1 \\ -1 & \text{if more than } k/2 \text{ elements of } Y \text{ are } -1. \end{cases}$$

Then for a data set $X$ where each element $x_i \in X$ has an associated label $y_i \in \{-1, +1\}$, define a $k$-nearest neighbor function $\phi_{X,k}(z)$ that returns the $k$ points in $X$ which are closest to a query point $z$. Next let sign report $y_i$ for any input point $x_i$; for a set of inputs $x_i$, it returns the set of values $y_i$.

Finally, the $k$-NN classifier is

$$g(z) = \mathsf{maj}(\mathsf{sign}(\phi_{X,k}(z))).$$

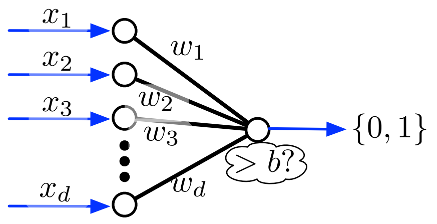
That is, it finds the $k$-nearest neighbors of query point $z$, and considers all of the class labels of those points, and returns the majority vote of those labels.

A query point $z$ near many other positive points will almost surely return $+1$, and symmetrically for negative points. This classifier works surprisingly well for many problems but relies on a good choice of distance function to define $\phi_{X,k}$.

Unfortunately, the model for the classifier depends on all of $X$. So it may take a long time to evaluate on a large data set $X$. In contrast the functions $g$ for all methods above take $O(d)$ time to evaluate for points in $\mathbb{R}^d$, and thus are very efficient.

## 9.5 Neural Networks

A *neural network* is a learning algorithm intuitively based on how a neuron works in the brain. A neuron takes in a set of inputs $x = (x_1, x_2, \ldots, x_d) \in \mathbb{R}^d$, weights each input by a corresponding scalar $w = (w_1, w_2, \ldots, w_d)$ and "fires" a signal if the total weight $\sum_{i=1}^d w_i x_i$ is greater than some threshold $b$.



$$\sum_{j=1}^{d} w_i \cdot x_i - b = \langle x, w \rangle - b > 0?$$

A neural network, is then just a network or graph of these neurons. Typically, these are arranged in layers. In the first layer, there may be $d$ input values $x_1, x_2, \ldots, x_d$. These may provide the input to $t$ neurons (each neuron might use fewer than all inputs). Each neuron produces an output $y_1, y_2, \ldots, y_t$. These outputs then serve as the input to the second layer, and so on.

Once the connections are determined, then the goal is to learn the weights on each neuron so that for a given input, a final neuron fires if the input satisfies some pattern (e.g., the input are pixels to a picture, and it fires if the picture contains a car). This is theorized to be "loosely" how the human brain works.

Given a data set $X$ with labeled data points $(x, y) \in X$ (with $x \in \mathbb{R}^d$ and $y \in \{-1, +1\}$), we already know how to train a single neuron so for input $x$ it tends to fire if $y = 1$ and not fire if $y = -1$. It is just a linear classifier. So, we can use the perceptron algorithm, or gradient descent with a well-chosen loss function!

However, for neural networks to attain more power than simple linear classifiers, they need to be at least two layers. Many amazing advances have come from so-called "deep neural networks" or "deep learning" or "deep nets" which are neural networks with many layers (say 20 or more). For these networks, the perceptron algorithm no longer works since it does not properly propagate across layers. However, a version of gradient descent called back-propagation can be used. Getting deep nets to work can be quite finicky. Their optimization function is not convex, and without various training tricks, it can be very difficult to find a good global set of weights. (The full details of this approach is well beyond the scope of this class.)